Submitted, 2017 Hybrid Systems: Communications and Control (HSCC)
http://www.cds.caltech.edu/~murray/preprints/ari+17-hscc_s.pdf

Fast Automatic Verification of Large-Scale Systems with Lookup Tables

Nikos Arichega Sumanth Dathathri Shashank Vernekar Sicun Gao
Shin'Ichi Shiraishi Richard M. Murray

ABSTRACT

Modern safety-critical systems are difficult to formally verify, largely due to their large scale. In particular, the widespread use of lookup tables in embedded systems across diverse industries, such as aeronautics and automotive systems, create a critical obstacle to the scalability of formal verification. This paper presents a novel approach for the formal verification of large-scale systems with lookup tables. We use a learning-based technique to automatically learn abstractions of the lookup tables and use the abstractions to then prove the desired property. If the verification fails, we propose a falsification heuristic to search for a violation of the specification. In contrast with previous work on lookup table verification, our technique is completely automatic, making it ideal for deployment in a production environment. To our knowledge, our approach is the only technique that can automatically verify large-scale systems lookup with tables.

We illustrate the effectiveness of our technique on a benchmark which cannot be handled by the commercial tool SLDV, and we demonstrate the performance improvement provided by our technique.

1. INTRODUCTION

Cyberphysical systems are growing in scale and complexity, and are being deployed for safety-critical applications as well as applications that require high performance and quality of service. These systems, used for in diverse applications such as aeronautics and driver-assistance features, require the highest level of assurance. Existing formal verification techniques, however, cannot scale to the size of even the smallest components, which we estimate would require millions of years on machines with millions of cores.

Based on our experience with industrial systems, we have observed that the widespread use of lookup tables creates a critical obstacle to scalability of formal methods. Lookup tables are an important and irreplaceable element of modern engineering design.

Lookup tables serve many critical roles in system design. Some are used to model severely nonlinear physical components—for which no satisfactory equational model exists. Others serve as control laws in cases where no traditional control design method delivers the required performance for these complex components. Others still serve as arithmetic shortcuts to quickly compute complex functions, such as trigonometrics, in embedded systems with timing constraints.

Lookup-tables are a challenge for traditional verification techniques because each entry of the lookup table must be treated as a separate case. If the system under analysis contains a large number of cascaded lookup tables, the number of proof cases grows exponentially, quickly outstripping the ability of a supercomputer to deliver timely verification results as part of a product-development cycle.

We estimate that a small component in a production system contains over 10^{50} total proof cases, which would require an estimated 10^{34} years on a machine with one million cores, assuming 0.01 seconds per proof case.

We propose a novel technique to improve the scalability of formal verification techniques for systems with lookup tables. First, we automatially learn an abstraction for each lookup table by treating the table data as training data. Our specially designed learning procedure learns a conservative overapproximation of the function implemented by the lookup table. This overapproximating abstraction can then be used to prove the desired property with an automatic SMT solver.

If the proof attempt fails, an SMT solver returns a *candidate counterexample* or a specific case in which the abstractions hold but the desired specification is violated. In this case, there are two possibilities, either the model is incorrect or the abstractions are inadequate.

The first possibility is that the system has a *true counterexample*, a condition that violates the desired

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

specification, but the case returned by the SMT solver is not necessarily a true counterexample. To address this first situation, we present a technique to search for a true counterexample based on examining the lookup table entries near the candidate counterexample. As a result, our abstractions can also be used to guide the search for a counterexample, yielding both verification and falsification functionality.

The second possibility is that the abstractions are too coarse. In this case, we describe an iterative refinement procedure which increases the fidelity of the abstractions, while incurring the cost of higher complexity abstractions.

We have implemented our abstraction, verification, and falsification techniques in the tool 'Osiris'. Osiris makes use of a user-extensible and modifiable library of templates. This extensibility is useful for engineers working in different domains, since the lookup tables that arise in different applications may be abstracted well by domain-specific abstraction templates.

We illustrate the performance of our approach on an adaptive cruise control benchmark with a monitor that tries to detect dangerous conditions. This benchmark could not be verified with the commercial tool Simulink Design Verifier (SLDV) nor with the SMT solver z3.

The paper is structured as follows. Section 2 describes related work, Section 3 provides background on lookup tables and SMT verification approaches, and 4 explains our problem statement. Section 5 describes our technique for computing abstractions, as well as how we use them for verification and falsification, and how we refine them if the specification cannot be proved nor falsified. Section 6 describes our tool, Section 7 presents our case study, and Sections 8 and 9 conclude and describe directions for future work.

2. RELATED WORK

To the best of our knowledge, no technique exists that can automatically verify large-scale systems with lookup tables.

The work of [14] uses a user-assisted mechanical theorem prover to prove safety of a large-scale aircraft collision avoidance system, which includes a large lookup table. The work of [14], however, relies on a human user to provide insight and manually reduce the system to simpler forms, until it is possible to derive input-output conditions on the lookup table to guarantee safety. This technique works top-down, starting from an overall system specification and decomposed with user assistance to a specification on the lookup table itself. This technique would be difficult to apply in a scenario with multiple cascaded lookup tables, since it would be difficult to decompose the high-level specification into obligations for each table, which would require the computationally infeasible task of propagating logical formulas through the tables.

In contrast, our technique works bottom-up, treating the lookup table as training data for an automatic learning procedure, which learns an abstraction of the lookup table. This abstraction is then used as part of an SMT query to check that the system specification is satisfied.

3. BACKGROUND

Lookup Tables.

Informally, a lookup table is a function defined by a table of input and output values. A lookup table maps certain points of its input space, called *breakpoints*, to values prescribed by a given table, such as the one shown in Table 1. The output of the function for values that do not appear in the table are computed by multi-linear interpolation if they are contained in the range of the breakpoints, and by saturation otherwise.

$x_1^{(1)}$	$\dots$	$x_i^{(1)}$	$\dots$	$x_n^{(1)}$	$y^{(1)}$
$\vdots$		$\vdots$		$\vdots$	$\vdots$
$x_1^{(j)}$	$\dots$	$x_i^{(j)}$	$\dots$	$x_n^{(j)}$	$y^{(j)}$
$\vdots$		$\vdots$		$\vdots$	$\vdots$
$x_1^{(m)}$	$\dots$	$x_i^{(m)}$	$\dots$	$x_n^{(m)}$	$y^{(m)}$

Table 1: Lookup Table with n inputs and m breakpoints

Formally, an n-dimensional lookup table with m-breakpoints is a function $\lambda : \mathbb{R}^n \to \mathbb{R}$, such that

1. for each breakpoint $(x^{(k)}, y^{(k)})$ $(k \in \{1, \dots, m\})$ that appears in the table, $\lambda(x^{(k)}) = y^{(k)}$, and
2. for every point $x \in \mathbb{R}^n$ that does not appear in the table,
   (a) if each component x_i is contained in the range of the lookup table, i.e. $\min_k(x_i^{(k)}) \leq x_i \leq \max_k(x_i^{(k)})$ for each $i \in \{1, \dots, n\}$, then $\lambda(x)$ is given by some interpolation function `interp`.
   (b) otherwise, $\lambda(x)$ is given by some extrapolation scheme `extrap`.

Our approach is general, and can be applied to any interpolation and extrapolation functions. However, in our case study, we will interpolate the lookup table by the multilinear interpolation formula described in [5]. For n dimensions, we will use the notation

$$\texttt{multiLinInterp}_n((x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), x)$$

to mean the n-dimensional interpolation function between points $(x^{(1)}, y^{(1)})$ and $(x^{(2)}, y^{(2)})$, evaluated at x. For simplicity, we will not extrapolate the lookup tables in our case study and simply assume that the range of interest is restricted to the range of the lookup tables.

Lookup tables as logical formulas.

Our technique relies on the ability to encode the system and its specification into first-order logic. An n-dimensional, m-breakpoint lookup table can be encoded as a first-order logical formula as follows.

Consider, for example, a two-dimensional lookup table with m breakpoints. The$k$-th breakpoint can be encoded by the following logical formula when $k = 1, \ldots, m-1$.

$$b_k \equiv x_1^{(k)} \leq x_1 \leq x_1^{(k+1)} \ \wedge \ x_2^{(k)} \leq x_2 \leq x_2^{(k+1)} \rightarrow \\ y = \texttt{multiLinInterp}_2((x^{(k)}, y^{(k)}), (x^{(k+1)}, y^{(k+1)}), x)$$

The vector x is the input of the lookup table, and x_1 and x_2 are its components. The function $\texttt{multiLinInterp}_2$ is bilinear interpolation.

The overall lookup table can be expressed by the conjunction of the logical formulas for the breakpoints.

$$L \equiv \bigwedge_{k=1}^{m} b_k \tag{1}$$

Satisfiability Modulo Theories.

Let $\Gamma(x)$ be a set of logical formulas with a vector of free variables x, and suppose x takes values in $\mathbb{R}^n$. The problem of *satisfiability modulo theory of the reals* is to find a point $r \in \mathbb{R}^n$ such that the logical formula $\Gamma(r)$ is true, or prove that none exists. In this case we say that r *satisfies* Γ.

Solvers exist that can solve the problem of satisfiability modulo theory of the reals for *subsets* of first-order logic, such as logical constraints that contain only polynomial functions over the reals[7]. The general problem is referred to as *satisfiability modulo theories* (SMT), since these solvers frequently support logical formulas over other sets, such as naturals or floating point numbers. Other solvers support transcendental functions, but relax the problem to finding an approximate solution or proving that not even an approximate solution exists [11].

4. PROBLEM FORMULATION

In this paper, we consider the problem of proving input-output properties of controllers. Our work could be extended to closed-loop properties of a control system, by applying our abstraction technique to the lookup tables, and then using a specialized solver to reason about the continuous portion of the dynamics. For example, bounded-time properties could be handled by a tool such as [12, 10, 8]. Unbounded-time properties would need to be supplemented by an automated invariant-guessing heuristic, such as [15].

Our approach can be applied to controller models that are given in first-order logic. In industry, controllers are frequently modeled either as imperative programs (for example in C), or as signal-flow models, e.g. Simulink.

In the case of imperative languages, Dynamic Logic provides a generic framework for translating common imperative control structures into first-order logic [13].

Numerous semantics have been proposed for signal-flow languages such as Simulink [16, 4] and Lustre. For our purposes, it suffices that the selected semantics should result in first-order logical constraints. We assume one of these existing semantic interpretations has already been used to translate the model appropriately, and in our case study we perform the translation manually.

Regardless of the original format of the controller, we assume that it has been translated to a set of logical constraints $\Sigma(x)$, not including any lookup tables, where x is the vector of *all* variables that occur in the system, including inputs, outputs, and intermediate assignment variables. We handle the lookup tables separately, and assume that each lookup table, indexed by i has been encoded as the first-order logic formula $L_i(x)$ as described in 3. Similarly, we assume that the specification is given as a first-order formula $S(x)$.

Then, the problem is to determine whether there exists a value of the variables x that

1. satisfies the model constraints $\Sigma(x)$, i.e., the values are related to each other according to the structure of the model;
2. satisfies each L_i, i.e., the values are related to each other in a way that satisfies the mapping produced by the lookup tables; and
3. does not satisfy the specification $S(x)$, i.e., it is an erroneous condition.

To check for the existence of this kind of erroneous condition, we can use an SMT solver to check the satisfiability of the following logical formula, assuming the number of lookup tables in the model is N.

$$\left(\bigwedge_{i=1}^{N} L_i \right) \wedge \Sigma(x) \wedge \neg S(x)$$

The key obstacle is that the lookup table formulas L_i are large, and existing techniques cannot scale. Our approach is to generate an abstraction A_i for each L_i by using the lookup table data as training data to learn parameters in an abstraction template. As a result, the logical formula will be simplified, but the abstraction loses information. To address this, we provide a falsification heuristic that can help to find true counterexamples when the verification does not succeed.

5. COMPUTING ABSTRACTIONS

Our approach to improve scalability is to abstract the lookup tables by *functional intervals.* A functional interval is a function that for each argument $x \in \mathbb{R}^n$ returns a (closed) interval over R, $A(x) = [a(x), b(x)]$. We say that a functional interval $A(x)$ *abstracts* a lookup

table $L(x)$ over a set $S \subseteq \mathbb{R}^n$ if for every $x \in S$, $L(x) \in A(x)$.

A functional interval abstraction is an *overapproximation* of a lookup table, in the sense that a property that holds for all values in the interval $A(x)$ must also hold of $L(x)$, but not vice-versa. The abstraction loses precision, but provides a simplification if the functions $a(x)$ and $b(x)$ have a sufficiently simple structure.

As a result, a procedure to compute a functional interval abstraction must balance between two conflicting requirements. On the one hand, it should be as precise as possible, by keeping the size of the interval small for every x, but it must also have a simple arithmetic structure, preferably consisting of linear or low-order polynomial terms, so that proving that the desired property holds of the abstraction is as simple as possible.

To navigate these conflicting requirements, we first try to abstract the lookup tables with linear abstractions, and see if these simple abstractions are sufficient to prove the specification or to guide the search to a counterexample. If the simple, linear abstractions are insufficient, then we iteratively increase the complexity to two linear pieces, then to quadratic, etc. As we describe in Section 6, our tool uses a library of abstraction templates that are indexed by complexity, and iterates through them on each subsequent abstraction attempt.

Next, we describe our procedure for computing abstractions, which consists of first computing an approximation and then finding an offset that makes the approximation into an upper and lower bound for the lookup table function. Then, we will describe our approach to learn piecewise abstractions. We proceed to describe how these abstractions can be used to search for a counterexample when the specification cannot be proved on the first attempt, and finally we explain how we refine our abstractions when no counterexample was found by the falsification procedure.

Computing abstractions by approximation.

We will use a learning-based procedure to automatically compute a functional interval for each lookup table in the model. First, we fix a parametric template for a function that approximates the lookup table data, and then we will proceed to learn parameter values that allow the function to approximate the lookup table data. Next, we use bisection search to search for the smallest offset that can be added and subtracted from the approximation to yield upper and lower bounds for the lookup table function.

We begin by learning an approximation of the lookup table data. Formally, let $f(a, x)$ be a function parametrized by $a \in \mathbb{R}^p$, with the same domain and range as the lookup table function L. We solve a regression problem to find the value of the parameter vector a that minimizes the mean-squared error over the breakpoints of

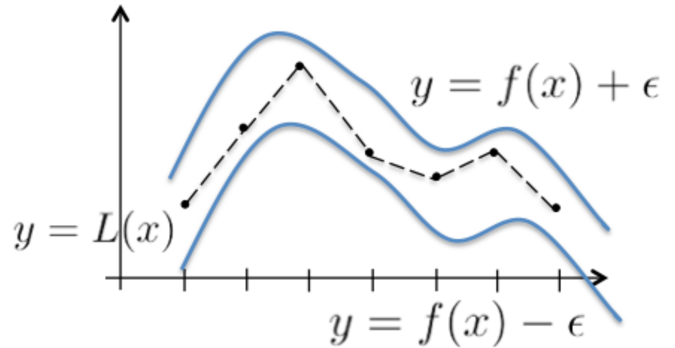

Figure 1: Lookup table function $L(x)$ abstracted by upper and lower bounding functions, obtained by shifting an approximation $f(x)$.

the lookup table.

$$\underset{a}{\text{minimize}} \sum_{i=1}^{k} (y^{(k)} - f(a, x^{(k)}))^2$$

Next, we use the approximation f to learn a functional interval. We begin by setting the offset to some initial value, eg. $\epsilon = 1$. Then, we use an SMT solver to check whether the lower and upper offset functions $f(x) - \epsilon$ and $f(x) + \epsilon$ are lower and upper bounds and for the lookup table function over all values in the range of interest $S \subseteq \mathbb{R}^n$. This is equivalent to checking the validity of the following logical formula with an SMT solver.

$$\forall x \in R \ . \ f(x) - \epsilon \leq L(x) \ \wedge \ L(x) \leq f(x) + \epsilon$$

Note that the expression for $L(x)$ contains the values of the breakpoints as well as the multilinear interpolation expressions in between the breakpoints of L.

If the validity check fails, i.e. the SMT solver is able to find an $x \in S$ such that the lookup table produces a value outside of the upper and lower bounds, we try again with a larger value of ϵ. If it succeeds, with this value as the upper cap (valid ϵ) and 0 (invalid ϵ) as the lower cap we do a bisection search to find the smallest value of ϵ(within some tolerance) such that the offset functions abstract the lookup table. This yields a functional interval,

$$A(x) = [f(x) - \epsilon, f(x) + \epsilon]$$

such that for all $x \in S$, $L(x) \in A(x)$. This relationship is illustrated in Figure 1 A bisection search is used to determine the optimal ϵ approximately as using a quantifier to directly determine the optimal ϵ, which turns out to be computationally expensive.

Approximation by piecewise functions.

Many lookup tables that appear in automotive software display sharp corners that cannot be well approximated by smooth functions. Our scheme learns func-

tional approximations that are piecewise smooth functions with simple arithmetic structure. The number of pieces in the approximation learned is much smaller than the number of interpolations in the lookup table.

We use the piecewise template of the following form.

$$f(a,x) = \begin{cases} f_1(a,x) & \text{if} \quad g_1(a,x) > 0 \\ f_2(a,x) & \text{if} \quad g_1(a,x) \leq 0 \ \wedge \ g_2(a,x) > 0 \\ & \vdots \\ f_k(a,x) & \quad \text{otherwise} \end{cases}$$

We cannot, however, directly train this template with lookup table data, since the function is not necessarily differentiable at the switching surfaces. Instead, we approximate the piecewise function by a smooth function. We begin by observing that the piecewise function can be represented as a sum of functions that are turned on or off by using a step function.

Let $s : \mathbb{R} \to \{0, 1\}$ denote the Heaviside step function.

$$s(x) = \begin{cases} 0 & \text{if} \quad x \leq 0 \\ 1 & \quad \text{otherwise} \end{cases}$$

Using the step function, we can represent the piecewise function of Eq. 5 by

$$\begin{aligned} f = & \, s(g_1) \cdot f_1 + (1 - s(g_2) \cdot f_2 + \ldots \\ & + \left(\prod_{i=1}^{r-1} (1 - s(g_i)) \right) \cdot s(g_r) f_r + \\ & + \left(\prod_{i=1}^{k-1} (1 - s(g_i)) \right) \cdot s(g_k) f_k . \end{aligned}$$

Note that f is possibly discontinous and non-differentiable. To enable learning the function with a gradient-based minimization technique, we approximate f with a continuous differentiable function by replacing the step function by the sigmoid function σ, which smoothly transitions from 0 to 1 as its argument crosses zero [6].

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$.f_a(x) = \sum_{i=1}^{k} f_i(x,a) \left(\prod_{k=1}^{i-1} (1 - \sigma(-g_k(x,a))) \right) \sigma(g_i(x,a))$$

5.1 Falsification

If the verification attempt does not succeed, it means that a value $x = \hat{x}$ was found such that the abstractions were satisfied, but the specification was falsified. This *candidate counterexample* is not necessarily a true counterexample, since a point that satisfies the abstractions may not satisfy the lookup tables.

However, this candidate counterexample serves as a flag of a region that may contain a true counterexample. It is sensible to search between the breakpoints that contain this counterexample, but note that this point may fall between different breakpoints in different lookup tables, which could potentially lead us to choose intervals from different lookup tables that are inconsistent with each other. To prevent this, instead of simply selecting the two breakpoints that contain the candidate counterexample, we select a small number r of the nearest breakpoints. See Figure 2 for an illustration of this mechanic. In our experiments, $r = 3$ or $r = 4$ are usually large enough to prevent inconsistent intervals.

Informally, we construct new lookup tables with only r entries each, and attempt to verify the same model with the reduced lookup tables, this time directly, without abstractions. If the verification succeeds, we know the candidate counterexample was spurious, and can repeat the procedure with a different candidate counterexample. If the verification fails, it provides a true counterexample which can be returned to the engineer as a design flaw that must be fixed.

Formally, let $x_j, \ldots, x_{j+n}$ be the n inputs of lookup table L_i. Then, consider the values of these variables in the candidate counterexample $\hat{x}_j, \ldots, \hat{x}_{j+n}$. We wish to extract the r nearest entries along each dimension—suppose they are $x_j^{(k)}, \ldots, x_j^{(k+r)}$ through $x_{j+n}^{(k)}, \ldots, x_{j+n}^{(k+r)}$. Then, construct a new lookup table $\hat{L}_i$ that contains only these breakpoints, and maps them to the same outputs as L_i. Finally, check satisfiability of the following logical formula.

$$\left(\bigwedge_i \hat{L}_i \right) \wedge \hat{\Sigma}(x) \wedge \neg S(x) \qquad (2)$$

If a satisfying instance is found, then that instance is a true counterexample of the original model. If no satisfying instance is found, then we can try the procedure with a different candidate counterexample that is at some minimum distance δ from $\hat{x}$. If there are no more candidate counterexamples at this minimum distance, we move on to the next step, which is to refine the abstractions and attempt verification again.

5.2 Abstraction refinement

When the SMT solver finds candidate counterexamples, unable to prove correctness, and the falsification procedure fails to find a true counterexample, we refine the abstractions and repeat the verification attempt. There are three basic mechanisms by which we refine abstractions: (1) increasing arithmetic complexity of the templates, (2) increasing the number of cases in a piecewise template, and (3) fitting the error of an abstraction.

Our tool implementation tries all three of these techniques at the same time, and keeps the technique that yields the approximation with lowest error.

The first two methods are simple. Increasing arithmetic complexity means moving from linear templates to quadratic templates, higher-order polynomials, or possibly transcendental functions if one is using an SMT

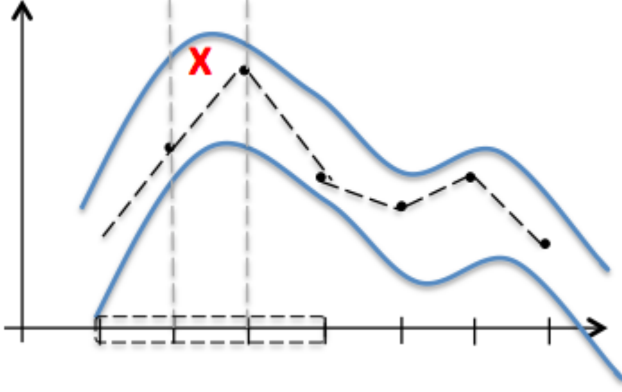

Figure 2: The red X represents a candidate counterexample. To search for a true counterexample, we construct a reduced table consisting of the four nearest breakpoints, which comprise the three intervals marked by the dotted rectangle

solver that supports such functions, such as [11]. Increasing the number of cases in a piecewise template means moving from a simple equational template to a template with two cases, or from two to three, etc.

Fitting the error of an abstraction.

This technique is loosely inspired by boosting methods for regression, where performance of a machine learning algorithm is improved by iteratively training on the residuals [9].

Informally, our idea is to replace a lookup table by its approximation plus a lookup table that computes the approximation error. Then, we compute an abstraction for this new error table, and proceed as before.

Suppose we have learned the abstraction $A_i(x) = [f_i(x)-\epsilon_i, f_i(x)+\epsilon_i]$ for the lookup table function $L_i(x)$. We can define the error lookup table function $E_i(x)$ as follows.

$$E_i(x) = f_i(x) - L_i(x) \tag{3}$$

Next, let y_i be the output of the lookup table L_i. We augment the logical constraints that define the system as follows.

$$\hat{\Sigma(x)} \equiv \Sigma(x) \wedge \bigwedge_i y_i = f_i(x)) \tag{4}$$

Next, we learn an abstraction $A_{E_i}(x)$ for $E_i(x)$. Finally, we check satisfiability of

$$\hat{\Sigma} \wedge \left(\bigwedge_i A_{E_i}(x) \right) \neg S(x). \tag{5}$$

Note that this error fitting procedure can be applied recursively to the error of the error table with its approximation, and in general the three techniques can be applied sequentially in any order. For this reason our tool performs all three at each refinement attempt and at each stage keeps the one that yields the smallest error.

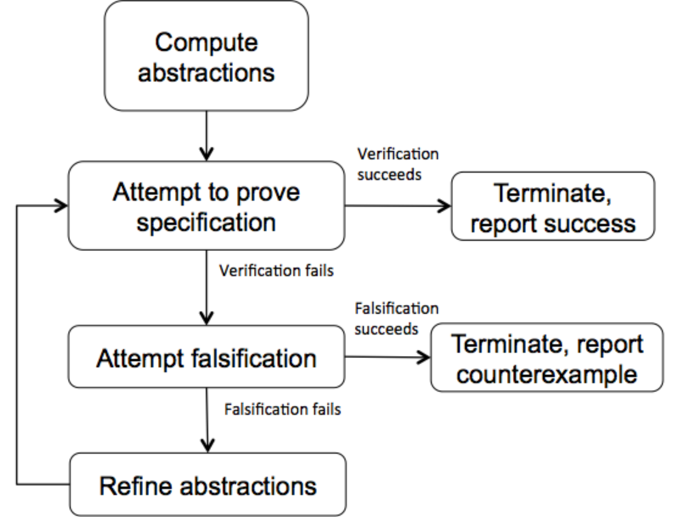

Figure 3: High-level view of Osiris

6. TOOL IMPLEMENTATION

We have implemented our technique in a tool called "Osiris". Figure 3 shows the overall flow of the tool.

Input.

The input to Osiris is a directory which contains

1. a file with extension `*.fmlas`, which lists the bounds on the inputs of the model, as well as calculations by system elements that are not lookup tables, and

2. for each lookup table, a *.m file, which contains a lookup table in standard Matlab syntax.

Learning abstractions with an extensible template library.

Osiris reads the templates for the approximating functions from an external library, which can be modified and extended by the user. The templates are sorted by complexity, starting by linear templates with no case split, followed by linear piecewise functions with two pieces and linear conditions, followed by quadratic functions with no case split, etc.

Osiris automatically starts working with a batch of the lowest complexity templates and computes approximations from them in parallel. All approximations are turned into abstractions by computing the smallest ϵ offset that produces upper and lower bounding function to the lookup table function. Checking the upper and lower bound properties is carried out with z3.

Since the learning procedure is not convex and therefore not guaranteed to converge to the same solution

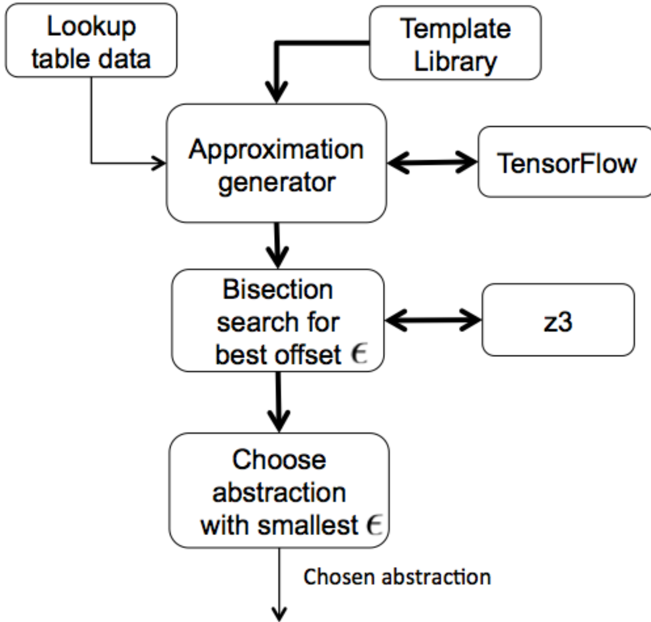

Figure 4: Flow of the abstraction generation procedure. Bold lines indicate multiple parallel instances.

from different initial parameter values, it is advantageous to have multiple copies of the same template at the same complexity index in the library. In this way, Osiris solves multiple instances in parallel with different initial parameter values. The final selection for lowest ϵ ensures that we will be able to keep the best abstraction. This flow is shown in Figure 4, where bold lines indicate multiple parallel instances.

Our library also has a section for domain-specific templates. We envision that future users of our technique will be interested in augmenting the library with templates from engine control, fluid dynamics, etc depending on the application. These domain-specific templates allow computing abstractions that roughly match the mathematics of the application.

Learning abstractions.

Osiris works through the template library in order of increasing complexity, using the machine learning tool TensorFlow to learn parameter values [3]. Osiris uses the SMT solver z3 in the bisection search procedure to search procedure to find the minimal offset ϵ that produces a true overapproximation of the lookup table function. Osiris also uses z3 to prove incrementally stronger relaxations of the desired specification, decreasing the parameter δ to find the strongest version of the property that can be proven ...

Proving specifications.

Once each abstraction A_i has been generated for each lookup table L_i ($i = 1, \dots, k$), Osiris forms the following logical formula.

$$A_1(x) \wedge \cdots \wedge A_k(x) \wedge \Sigma(x) \wedge \neg S(x)$$

Then, the SMT solver z3 is used to check for satisfiability. If the formula is not satisfiable, z3 has proven that there is no value that satisfies the abstractions and the model constraints but falsifies the specification. Since the abstractions overapproximate the lookup table functions, it follows that the system with the lookup table functions satisfies its specifications.

Conversely, if a violation x_c is found, this does not necessarily mean that the original system violates its specifications. For each lookup table L_i, Osiris finds nearest breakpoints in each lookup table.

Falsification.

7. CASE STUDY

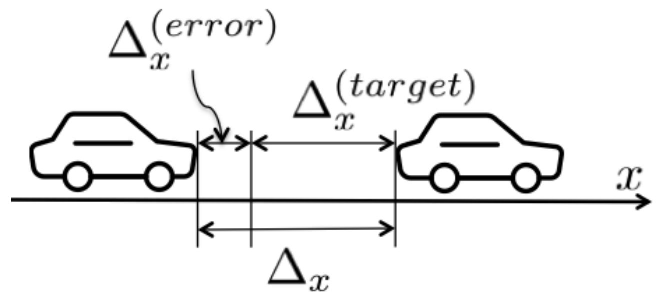

Figure 5: Diagram of adaptive cruise control scenario

For our case study, we consider an adaptive cruise controller along with an online monitor. When enabled, adaptive cruise control regulates the speed of the car so that a target speed is maintained, unless another car is detected at some distance in front, in which case the system tries to maintain a safe distance from the lead car, as shown in Figure 5. This controller takes as input the current speed of the car, the distance to the lead car, and the relative speed between the two cars.

The system consists of a cascade of three lookup tables, as shown in Figure 6. The inputs to the controller are s, the speed of the controlled car, Δ_x, the distance to the leading car, and Δ_v the relative speed of the two cars.

The first lookup table uses the current velocity s of the controlled car to determine a target set distance ($\Delta_x^{(target)}$) from the leading car. If the controlled car is moving fast, its braking distance will be larger, which requires that the controller choose a longer following distance. $\Delta_x^{(error)}$ is the difference between the target following distance and the chosen following distance, and the second lookup table uses $\Delta_x^{(error)}$ together with the relative velocity Δ_v to choose an acceleration. For the first two lookup tables, we have chosen breakpoints

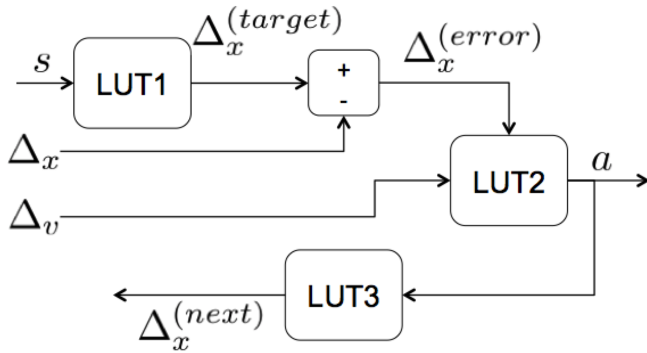

Figure 6: Signal-flow model of an ACC controller

that worked well in simulation. The third lookup table behaves as an online monitor. In practice, a monitor lookup table would be produced by recording observations of a physical component. For this example, the monitor was generated by computing the future distance between the two cars after 0.1 seconds, given the current distance, relative velocity, and chosen acceleration. This monitor assumes that the lead car will not change its velocity within the next 0.1 seconds.

The property we wish to prove is that the online monitor will never predict a future distance that is negative, i.e., it will never predict that the cars will crash. This does not mean that the closed-loop system with the real automotive dynamics will not crash, since that would require analyzing the continuous-time differential equations. However, industrial controllers are frequently equipped with online monitors that predict or prevent dangerous conditions, and checking that the controller satisfies its monitor is valuable, as it prevents any abnormal behavior as long as system integrity is preserved.

The first lookup table contains 21 breakpoints breakpoints, the second contains 1155 breakpoints and the third, monitor lookup table contains 385 of breakpoints. In total, the cascaded lookup tables produce $9,338,175$ proof cases.

We have implemented this system in Simulink, and we attempted to prove it with Simulink Design Verifier (SLDV), a commercial formal verification tool developed by The MathWorks [1]. However, SLDV could not verify this model. We conjecture that this is due to SLDV's present limitations at working with nonlinear functions [2]. Our model contains non-linear calculations in the interpolation of the 2D and 3D lookup tables, which are bilinear and trilinear, respectively.

We are in discussion with our legal department about making this benchmark public, and we hope it will be publicly available by the time this article appears in print. Since it is a Simulink model, our organization regards it as sensitive material, and a review is required.

We translated our model to first order logic to use an SMT solver to check validity of the specification. We do not include the logical formulas that represent the lookup table due to space constraints.

$$
\begin{aligned}
&0 \le \Delta_x \le 180\\
&-50 \le \Delta_v \le 50\\
&0 \le s \le 180\\
&\Delta_x^{(target)} = \mathtt{LUT}_1(s)\\
&\Delta_x^{(error)} = \Delta_x - \Delta_x^{(target)}\\
&a = \mathtt{LUT}_2(\Delta_x^{(error)}, \Delta_v)\\
&\Delta_x^{(next)} = \mathtt{LUT}_3(\Delta_x, \Delta_v, \Delta_x^{(next)})
\end{aligned}
$$

The constraints on Δ_x, Δ_v, and s are assumptions on the bounds of these inputs, and the system cannot be enabled if these bounds are not met. Similarly, commercial adaptive cruise control systems cannot be used if the speed of the controlled car is too slow.

We also attempted to verify this model by translating it into first-order logic constraints and using z3 to check for a violation of the specification, but z3 did not terminate after 48 hours.

When we ran this model in Osiris, a counterexample was found in 3 minutes and 30 seconds, as follows.

$$
\begin{aligned}
&s \mapsto 31.0\\
&a \mapsto -2.0\\
&\Delta_v \mapsto -4.0\\
&\Delta_x \mapsto 0.03125\\
&\Delta_x^{(error)} \mapsto -30.97\\
&\Delta_x^{(target)} \mapsto 31.0\\
&\Delta_x^{(next)} \mapsto -0.00865
\end{aligned}
$$

The meaning of this counterexample is that the cars start at a distance Δ_x of about 3 cm, with a relative velocity of $-4m/s$, i.e. the controlled car is moving $4m/s$ faster than the lead car. The controller tries to brake by applying a negative acceleration of $a = -2m/s^2$, but the situation is already too dangerous and the cars have a minor crash, with the controlled car being $0.8cm$ further than it should be.

To measure the runtime of our verification technique, we relaxed the specification too $\Delta_x^{(next)} \ge -1.76$. With this relaxed property, the monitor no longer tries to completely prevent collisions, but simply to reduce their severity. This relaxed property was provable in 20 minutes.

The case study computations were carried out on a machine with 8 cores and 132 GB of RAM.

7.1 Computed abstractions

The abstraction computed for LUT1 consists of a linear function, shifted above and below the lookup table data.

$$A_1 = [1.27s + 0.43 + 29.69, 1.27s + 0.43 - 29.69]$$

We have deliberately left the constants un-simplified.

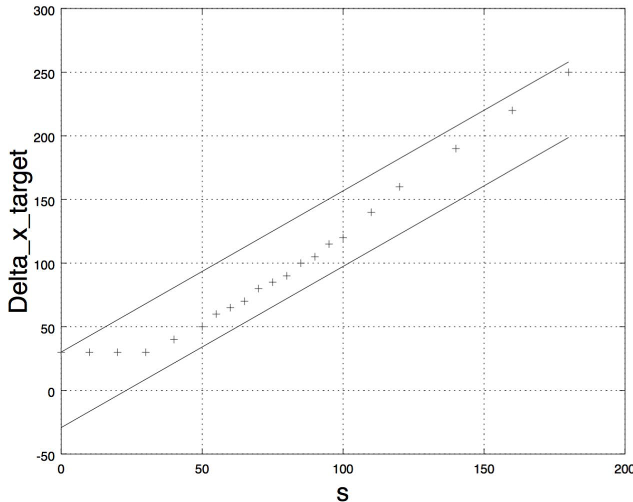

Figure 7: Plot of LUT 1 data and abstractions

The constant $\epsilon_1 = 29.69$ is useful because it represents the largest error between the abstraction and the lookup table itself. Thus, we can compare which lookup tables are being abstracted with more or less fidelity by looking at the value of ϵ.

The abstraction computed for LUT2 is a piecewise linear function, and has the form

$$A_2 = [f_2(x) + \epsilon_2, f_2(x) - \epsilon_2]$$

where

$$f_2 = \begin{cases} 0.024\Delta_x^{(error)} + 0.093\Delta_v - 0.508 & \text{if} \quad g_2(x) \geq 0 \\ 0.020\Delta_x^{(error)} + 0.093\Delta_v - 0.508 & \quad \text{otherwise} \end{cases}$$

and

$$g_2(x) = 0.321\Delta_x^{(error)} - 1.232\Delta_v - 1$$

The abstraction computed for LUT3 is also a piecewise linear function,

$$A_3 = [f_3(x) + \epsilon_3, f_3(x) - \epsilon_3]$$

where

$$f_3 = \begin{cases} f_3^{(1)}\text{if} & g_3(x) \geq 0 \\ f_3^{(2)} & \text{otherwise} \end{cases}$$

and

$$f_3^{(1)} = 0.997\Delta_x + 0.006\Delta_v - 0.003\Delta_x^{(next)} + 0.409$$
$$f_3^{(2)} = 1.593\Delta_x - 0.006\Delta_v - 0.059\Delta_x^{(next)} + 1.143$$
$$g_3(x) = 0.795\Delta_x - 0.222\Delta_v - 0.476\Delta_x^{(next)} + 0.503.$$

8. CONCLUSION

We have provided a technique to automatically compute abstractions of lookup tables. We treat the lookup table breakpoints as training data for a learning procedure, and learn an abstraction of the lookup table. Abstracting the lookup tables allows for fast, automatic verification of input-output properties of large-scale controllers with lookup tables. To the best of our knowledge, this is the only automatic technique for this purpose.

We have implemented our technique in the tool Osiris, which parses a set of logical constraints along with lookup tables represented as Matlab programs.

Osiris uses an extensible library of abstraction templates, sorted by complexity. Starting at the lowest level of complexity, Osiris attempts to generate abstractions to prove the desired specification, increasing the level of complexity after each failed attempt.

The extensible nature of our template library makes it easy for engineers in different application domains to add templates that may provide a good fit to the lookup tables that appear in their discipline.

9. FUTURE WORK

We have identified several directions for future work. We would like to generalize the form of the abstractions that we use, since our current abstractions are limited to upper and lower bounding functions. Also, we would like to explore the possibility of using a learning method with better theoretical guarantees (e.g. SVMs), since our current setup yields a non-convex optimization problem.

The current implementation of Osiris can only check single specifications. In the case when the specification does not hold, one can obtain insight by relaxing the specification. In future work, we would like to support sets of specifications, and to construct a lattice of relaxed specifications, which Osiris would iteratively attempt to prove. In this way, we would be able to provide the designer with a trade-off curve of which of the desired specifications are easier or more difficult to prove. We also plan to extend our template library as we gain experience with further case studies.

Osiris currently only supports static SMT tools as back-end solvers. In future work, we would like to extend the applicability of our technique to handle closed-loop control systems, which would require a hybrid model checking tool, or some kind of automatic invariant guessing heuristic.

Additionally, we would like to provide better feedback to the designer about which parts of the design seem to be most critical to satisfying the specifications. This would require an efficient way to quantify the expected improvement in the specification with respect to the improvement of each abstraction.

10. REFERENCES

[1] Simulink design verifier. http://www.mathworks.com/products/sldesignverifier/.

[2] Supported and unsupported blocks in simulink design verifier. https://www.mathworks.com/help/sldv/ug/simulink-block-support.html.

[3] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Josefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow. Technical report, Google Research, 2015.

[4] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of simulink/stateflow model to hybrid automata using graph transformations. *Electronic Notes in Theoretical Computer Science*, 2004.

[5] S. E. Z. Alan Weiser. A note on piecewise linear and multilinear table interpolation in many dimensions. *Mathematics of Computation*, 50(181):189–196, 1988.

[6] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, 1989.

[7] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, International Conference on Tools and Algorithms for the Construction and Analysis of Systems'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] A. Donzé. Breach, a tootool for verification and parameter synthesis of hybrid systems. In *Computer Aided Verification*, 2010.

[9] N. Duffy and D. Helmbold. Boosting methods for regression. *Machine Learning*, 47(2):153–200, 2002.

[10] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. SpaceEx: Scalable verification of hybrid systems. In *Int. Conf on Computer Aided Verification*, pages 379–395, July 2011.

[11] S. Gao, S. Kong, and E. M. Clarke. dReal: An SMT solver for nonlinear theories of the reals. In *Int. Conf. on Automated Deduction*, June 2013.

[12] S. Gao, S. Kong, and E. M. Clarke. Satisfiability modulo ODEs. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 105–112. IEEE, October 2013.

[13] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, Foundations of Computing Series, 2000.

[14] J. Jeannin, K. Ghorbal, Y. Kouskoulas, R. Gardner, A. Schmidt, E. Zawadzki, and A. Platzer. Formal verification of ACAS X, an industrial airborne collision avoidance system. In A. Girault and N. Guan, editors, *EMSOFT*, pages 127–136. IEEE Press, 2015.

[15] J. Kapinski, J. Deshmukh, S. Sankaranarayanan, and N. Aréchiga. Simulation-guided Lyapunov analysis for hybrid dynamical systems. In *Int. Conf. on Hybrid Systems: Computation and Control*, April 2014.

[16] A. Tiwari. Formal semantics and analysis methods for simulink stateflow models. Technical report, SRI International, 2002.